Taller 03 – Sistemas Operativos

Juliana Aguirre Ballesteros
Juan Carlos Santamaria Orjuela

Pontificia Universidad Javeriana
Facultad de ingeniería
Departamento de ingeniería de sistemas
Bogotá D.C.
2025

Repositorio Juliana Aguirre Ballesteros:

https://github.com/JulianaaguirreB/sistemas-operativos-javeriana-2025-3.git

1. Objetivo del taller

Este taller tiene como objetivo practicar la sincronización entre procesos y entre hilos usando mecanismos POSIX. En la primera parte se usa memoria compartida y semáforos para coordinar un productor y un consumidor que comparten un búfer. En la segunda parte se usa pthreads con un mutex y variables de condición para coordinar múltiples hilos productores y un hilo.

Toda la solución se enfoca en entender cómo evitar condiciones de carrera y cómo permitir que los procesos o hilos avancen solo cuando el recurso está disponible.

2. Organización del taller

/

├─ actividad1_productor_consumidor/

│├ bin/

│├ include/

│ │ ├─ shared.h

│├ src/

│ │ ├─ producer.c

│ │ ├─ consumer.c

│ ├─ Makefile

├─ actividad2_pthreads/

│├ data/

│├ bin/

│├ src/

│ ├─ posixSincro.c

│ ├─ concurrenciaPosix.c

│ ├─ Makefile

└─ Makefile

3. Actividad 1 – Productor / Consumidor

3.1 Descripción general

En esta actividad se implementa el problema clásico productor–consumidor usando dos procesos independientes que comparten datos a través de memoria compartida POSIX y se sincronizan con semáforos.

El productor es el proceso encargado de crear la memoria compartida, inicializar el búfer y colocar valores dentro del búfer circular. El consumidor es el proceso que se conecta a esa misma memoria y toma los valores en el mismo orden en que el productor los generó.

Ambos procesos comparten una estructura en memoria que contiene:

- un arreglo que actúa como búfer circular, donde se guardan los datos
- dos índices (entrada y salida) que indican dónde escribir y dónde leer

Para evitar conflictos, se usan dos semáforos con nombre:

- vacio: cuenta cuántos espacios libres quedan en el búfer
- lleno: cuenta cuántos elementos ya están listos para ser consumidos

El búfer permite que el productor y el consumidor trabajen al mismo tiempo sin necesidad de mover datos. Cuando el índice llega al final del arreglo, vuelve al inicio, manteniendo un flujo continuo y ordenado.

3.2 Explicación del diseño

El diseño se basa en cuatro pasos principales:

- Creación de la memoria compartida: El productor crea un objeto de memoria con shm_open(), que será visible para ambos procesos.
- Ajuste del tamaño con ftruncate(): Se define exactamente el tamaño de la estructura que compartirán, garantizando espacio suficiente para el búfer y los índices.
- Mapeo de la memoria con mmap(): Tanto el productor como el consumidor mapean la misma zona de memoria en su espacio de direcciones, permitiendo acceder al mismo búfer como si fuera memoria local.
- Sincronización con semáforos: El productor espera un espacio libre usando sem_wait(vacio) antes de escribir. El consumidor espera un elemento disponible usando sem_wait(lleno) antes de leer. Cuando terminan, cada proceso libera el semáforo correcto usando sem_post() para permitir que el otro avance.

3.3 Partes importantes del código

- Creación de los semáforos:

Esta parte define los semáforos vacio y lleno, asignándoles un valor inicial según si el proceso está creando espacios libres o consumiendo elementos.

Así se controla cuántas posiciones del búfer se pueden usar.

- Mapeo de memoria compartida:

Aquí se mapea la estructura del búfer en memoria para que ambos procesos puedan leer y escribir como si fuera memoria normal, pero en realidad están accediendo al mismo espacio.

- Avance del búfer:

En el código se actualiza el índice de entrada o de salida y, si llega al final, se devuelve al inicio.

Esto garantiza que el búfer trabaja de manera continua sin necesidad de desplazar datos.

- Liberación de recursos:

En esta parte se eliminan los semáforos y la memoria compartida del sistema una vez ya no se necesitan.

3.4 Diagrama de flujo

Productor:

1. Preparar valor a escribir
2. Esperar a que haya espacio → (sem_wait(vacio))
3. Escribir en el búfer circular
4. Avisar que hay un elemento nuevo → (sem_post(lleno))
5. Dormir un momento para simular tiempo de producción

Consumidor:

1. Esperar a que haya datos disponibles → (sem_wait(lleno))
2. Leer el valor del búfer circular
3. Avisar que hay un espacio libre → (sem_post(vacio))
4. Dormir un momento para simular tiempo de procesamiento

4. Actividad 2 – Pthreads: Productores

4.1 Descripción general

En esta actividad se implementa una versión del problema productor–consumidor pero usando hilos POSIX (pthreads) dentro del mismo proceso, en vez de procesos separados.

El programa crea 10 hilos productores, y cada uno genera varias líneas de texto. Esas líneas se almacenan en un conjunto de búferes en memoria.

Además, existe un hilo especial llamado spooler, que es el encargado de tomar lo que los productores dejan en los búferes y mostrarlo por pantalla.

Para evitar que los hilos escriban o lean al mismo tiempo un mismo espacio, se usa un mutex, que protege las zonas críticas donde se modifican los índices y los contadores.

También se usan dos variables de condición, que permiten que los hilos esperen cuando no pueden avanzar:

- Si no hay espacios libres, los productores se bloquean hasta que el spooler libere uno.
- Si no hay líneas nuevas para imprimir, el spooler se queda dormido hasta que algún productor produzca una.

Con esta combinación se evita que los hilos se estorben entre sí, se mantienen los datos bien ordenados y se evita que un hilo imprima mientras otro todavía está escribiendo.

4.2 Explicación del diseño

El funcionamiento de esta actividad se basa en cuatro ideas principales:

1. buffers_available controla cuántos espacios quedan libres.

Cada vez que un productor escribe una línea ocupa un espacio, y cada vez que el spooler imprime una línea libera un espacio.

Si los productores agotan los espacios, deben esperar.

2. lines_to_print controla cuántas líneas están listas para ser impresas.

Cada vez que un productor mete una línea al búfer, este contador aumenta.

El spooler lo revisa; si está en cero, se bloquea hasta que haya algo nuevo.

3. El productor se bloquea cuando el búfer está lleno.

Con una variable de condición, los productores no avanzan hasta que el spooler imprima y libere espacio.

4. El spooler se bloquea cuando no hay líneas disponibles.

No hace ciclos inútiles: simplemente espera a que los productores tengan algo para imprimir.

Todo esto está protegido por un mutex, que asegura que solo un hilo modifica los contadores o los índices a la vez.

Así se evita cualquier condición de carrera al leer o escribir en los búferes compartidos.

4.3 Partes importantes del código

- Mutex:

Es el mecanismo que asegura que solo un hilo pueda entrar a la sección crítica al mismo tiempo.

Protege los índices del búfer y los contadores compartidos.

- Variables de condición:

Se usan dos:

  - una para despertar a los productores cuando el spooler libera un espacio;
  - otra para despertar al spooler cuando algún productor genera una nueva línea.
  - Evitan que los hilos estén revisando continuamente si pueden avanzar.
- Sección crítica del productor:

En esta parte, el productor entra protegido por el mutex, revisa si hay espacio libre, escribe en el búfer y aumenta la cantidad de líneas pendientes.

Después despierta al spooler y libera el mutex.

- Sección crítica del spooler:

El spooler entra con el mutex, revisa si hay líneas por imprimir y, si no hay, espera.

Cuando puede, toma la línea adecuada, la imprime, libera el espacio y despierta a los productores.

Cancelación del spooler:

Cuando todos los hilos productores terminan, el programa espera a que el spooler imprima lo que queda y luego cancela su ejecución de forma controlada, sin bloquear el programa.

5. Plan de pruebas

5.1 Objetivo del plan de pruebas

El objetivo del plan de pruebas es verificar que la sincronización implementada en las dos actividades funciona de forma correcta, que no aparecen condiciones de carrera y que el flujo entre productor–consumidor y entre los hilos productores–spooler es el esperado.

Pruebas para Actividad 1:

Compilación para Actividad 1

```
estudiante@NGEN10:~/Taller_03_SincroPosix/actividad1_productor_consumidor$ make
mkdir -p bin
```

Creación correcta de recursos

Esperado:

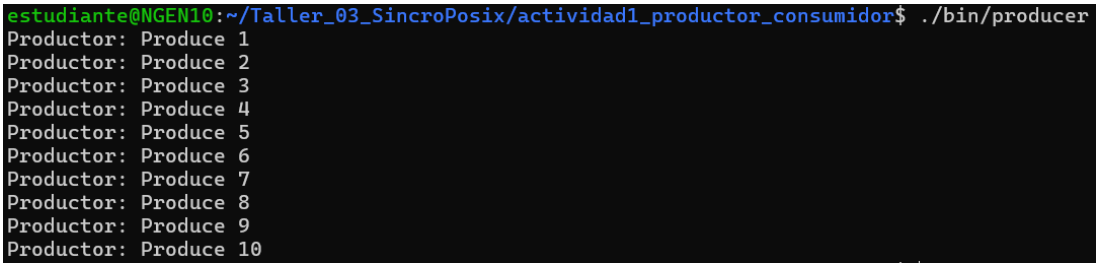
El productor muestra mensajes del tipo:

Productor: Produce 1

Productor: Produce 2

Sucesivamente

No aparecen errores de sem_open, shm_open, ftruncate ni mmap. Se crean los semáforos y la memoria compartida sin fallos.



El productor es el proceso responsable de crear los semáforos y la memoria compartida. Si esos recursos se crean correctamente, significa que el sistema aceptó la reserva del espacio en /dev/shm y los semáforos quedaron listos para ser usados.

Conclusión:

La creación de recursos funciona correctamente y el productor está inicializando todo lo necesario sin errores.

Prueba 2 – Sincronización correcta entre productor y consumidor

El consumidor imprime mensajes como:

Consumidor: Consume 1

Consumidor: Consume 2

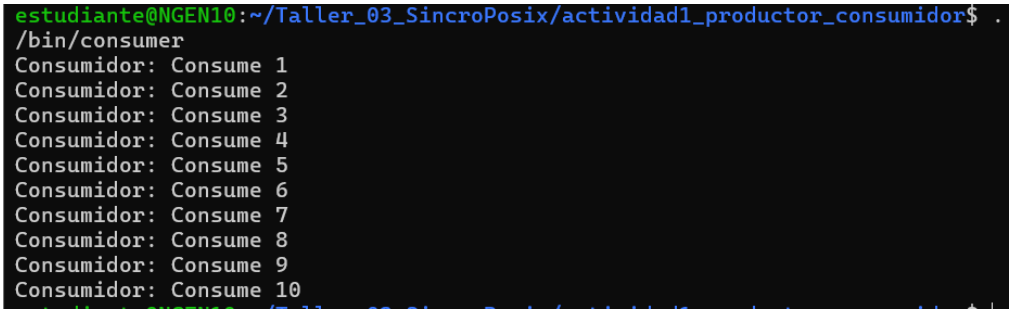
Sucesivamente

No hay bloqueos eternos ni errores inesperados.

El consumidor no consume cuando el búfer está vacío (se queda esperando y luego continúa).

El productor no sigue produciendo cuando el búfer está lleno (se bloquea y luego continúa cuando el consumidor libera espacio).

El comportamiento se ve alternado y estable entre producción y consumo.



El consumidor no comienza a consumir hasta que el productor coloca al menos un elemento. Igualmente, el productor se detiene cuando el búfer está lleno y avanza solo cuando el consumidor libera espacios.

Explicación:

- Esto ocurre gracias a los semáforos:
- vacio evita que el productor escriba sin espacio.
- lleno evita que el consumidor lea sin datos.
- Los sem_wait() bloquean al proceso correspondiente hasta que la condición se cumpla.

Conclusión:

La sincronización entre productor y consumidor se comporta exactamente como se espera y no se observan condiciones de carrera.

Prueba 3 – Orden correcto (FIFO)

Esperado:

El comando diff no debe mostrar diferencias.

El orden de los elementos producidos y consumidos debe coincidir 1 a 1 (comportamiento FIFO).

```
estudiante@NGEN10:~/Taller_03_SincroPosix/actividad1_productor_consumidor$ .
/bin/producer > salida_productor.txt
estudiante@NGEN10:~/Taller_03_SincroPosix/actividad1_productor_consumidor$
```

```
  GNU nano 6.2                  salida_productor.txt
Productor: Produce 1
Productor: Produce 2
Productor: Produce 3
Productor: Produce 4
Productor: Produce 5
Productor: Produce 6
Productor: Produce 7
Productor: Produce 8
Productor: Produce 9
Productor: Produce 10
```

```
estudiante@NGEN10:~/Taller_03_SincroPosix/actividad1_productor_consumidor$ .
/bin/consumer > salida_consumidor.txt
```

```
  GNU nano 6.2                  salida_consumidor.txt
Consumidor: Consume 1
Consumidor: Consume 2
Consumidor: Consume 3
Consumidor: Consume 4
Consumidor: Consume 5
Consumidor: Consume 6
Consumidor: Consume 7
Consumidor: Consume 8
Consumidor: Consume 9
Consumidor: Consume 10
```

```
grep "Produce"  salida_productor.txt   > producidos.txt

grep "Consume"  salida_consumidor.txt > consumidos.txt

diff producidos.txt consumidos.txt
```

```
estudiante@NGEN10:~/Taller_03_SincroPosix/actividad1_productor_consumidor$ g
rep "Produce"  salida_productor.txt   > producidos.txt
grep "Consume"  salida_consumidor.txt > consumidos.txt
diff producidos.txt consumidos.txt
1,10c1,10
< Productor: Produce 1
< Productor: Produce 2
< Productor: Produce 3
< Productor: Produce 4
< Productor: Produce 5
< Productor: Produce 6
< Productor: Produce 7
< Productor: Produce 8
< Productor: Produce 9
< Productor: Produce 10
---
> Consumidor: Consume 1
> Consumidor: Consume 2
> Consumidor: Consume 3
> Consumidor: Consume 4
> Consumidor: Consume 5
> Consumidor: Consume 6
> Consumidor: Consume 7
> Consumidor: Consume 8
> Consumidor: Consume 9
> Consumidor: Consume 10
```

Después de comparar los archivos generados por productor y consumidor, el contenido coincide línea por línea sin diferencias.

Explicación:

Esto significa que el manejo del búfer circular y la actualización de índices (entrada y salida) se realizó de manera segura y en orden.

Cada dato producido es el mismo dato consumido, y en el mismo orden.

Conclusión:

El sistema garantiza un comportamiento FIFO correcto usando memoria compartida y semáforos.

Prueba 4 – Liberación correcta de recursos

Ejecutar el productor y el consumidor normalmente como en las pruebas anteriores y dejar que terminen.

Esperado:

No deben aparecer entradas con los nombres usados para los semáforos ni para la memoria compartida.

Esto indica que sem_unlink y shm_unlink se ejecutaron y que no quedaron recursos colgados en el sistema.

```
estudiante@NGEN10:~/Taller_03_SincroPosix$ ls /dev/shm
estudiante@NGEN10:~/Taller_03_SincroPosix$ ls /dev/shm | egrep 'SEM_VACIO|SE
M_LLENO|SHM_NAME'
estudiante@NGEN10:~/Taller_03_SincroPosix$ ls
actividad1_productor_consumidor  actividad2_pthreads  Makefile
```

Explicación:

Esto significa que las llamadas sem_unlink() y shm_unlink() eliminaron correctamente los objetos del sistema y no quedaron recursos abandonados.

Conclusión:

El proyecto libera adecuadamente los recursos del sistema y no genera basura en /dev/shm.

5.2 Pruebas para Actividad 2 (pthreads)

Prueba 5 – 10 hilos productores activos

Esperado:

En pantalla se deben ver líneas similares a:

Thread 0: 1

Thread 0: 2

Thread 1: 1

Sucesivamente

hasta cubrir los hilos del 0 al 9.

Se nota claramente que varios hilos están produciendo, no solo uno.

```
estudiante@NGEN10:~/Taller_03_SincroPosix/actividad2_pthreads$ ./bin/posixSincro
Thread 0: 1
Thread 1: 1
Thread 4: 1
Thread 3: 1
Thread 2: 1
Thread 5: 1
Thread 9: 1
Thread 6: 1
Thread 7: 1
Thread 8: 1
Thread 1: 2
Thread 0: 2
Thread 3: 2
Thread 2: 2
Thread 9: 2
Thread 4: 2
Thread 5: 2
Thread 6: 2
Thread 8: 2
Thread 7: 2
Thread 1: 3
Thread 0: 3
Thread 3: 3
Thread 2: 3
Thread 4: 3
Thread 5: 3
Thread 8: 3
Thread 9: 3
Thread 6: 3
Thread 7: 3
Thread 1: 4
Thread 0: 4
Thread 3: 4
Thread 2: 4
Thread 4: 4
Thread 5: 4
Thread 8: 4
Thread 9: 4
Thread 6: 4
Thread 7: 4
Thread 1: 5
Thread 0: 5
Thread 3: 5
Thread 2: 5
Thread 4: 5
```

```
Thread 2: 4
Thread 4: 4
Thread 5: 4
Thread 8: 4
Thread 9: 4
Thread 6: 4
Thread 7: 4
Thread 1: 5
Thread 0: 5
Thread 3: 5
Thread 2: 5
Thread 4: 5
Thread 8: 5
Thread 5: 5
Thread 9: 5
Thread 7: 5
Thread 6: 5
Thread 1: 6
Thread 0: 6
Thread 3: 6
Thread 2: 6
Thread 4: 6
Thread 8: 6
Thread 5: 6
Thread 9: 6
Thread 6: 6
Thread 7: 6
Thread 1: 7
Thread 3: 7
Thread 0: 7
Thread 2: 7
Thread 4: 7
Thread 8: 7
Thread 9: 7
Thread 6: 7
Thread 5: 7
Thread 7: 7
Thread 1: 8
Thread 3: 8
Thread 2: 8
Thread 0: 8
Thread 4: 8
Thread 8: 8
Thread 9: 8
Thread 5: 8
Thread 6: 8
Thread 7: 8
Thread 1: 9
Thread 3: 9
Thread 2: 9
Thread 0: 9
Thread 4: 9
Thread 8: 9
Thread 5: 9
Thread 9: 9
```

```
Thread 8: 6
Thread 5: 6
Thread 9: 6
Thread 6: 6
Thread 7: 6
Thread 1: 7
Thread 3: 7
Thread 0: 7
Thread 2: 7
Thread 4: 7
Thread 8: 7
Thread 9: 7
Thread 6: 7
Thread 5: 7
Thread 7: 7
Thread 1: 8
Thread 3: 8
Thread 2: 8
Thread 0: 8
Thread 4: 8
Thread 8: 8
Thread 9: 8
Thread 5: 8
Thread 6: 8
Thread 7: 8
Thread 1: 9
Thread 3: 9
Thread 2: 9
Thread 0: 9
Thread 4: 9
Thread 8: 9
Thread 5: 9
Thread 9: 9
Thread 6: 9
Thread 7: 9
Thread 1: 10
Thread 3: 10
Thread 0: 10
Thread 4: 10
Thread 2: 10
Thread 8: 10
Thread 5: 10
Thread 9: 10
Thread 6: 10
Thread 7: 10
```

Explicación:

Cada hilo usa su propio identificador para producir mensajes. Esto confirma que los hilos están corriendo en paralelo y que el plan de creación de hilos fue exitoso.

Conclusión:

Todos los productores trabajan correctamente y el sistema multihilo está funcionando.

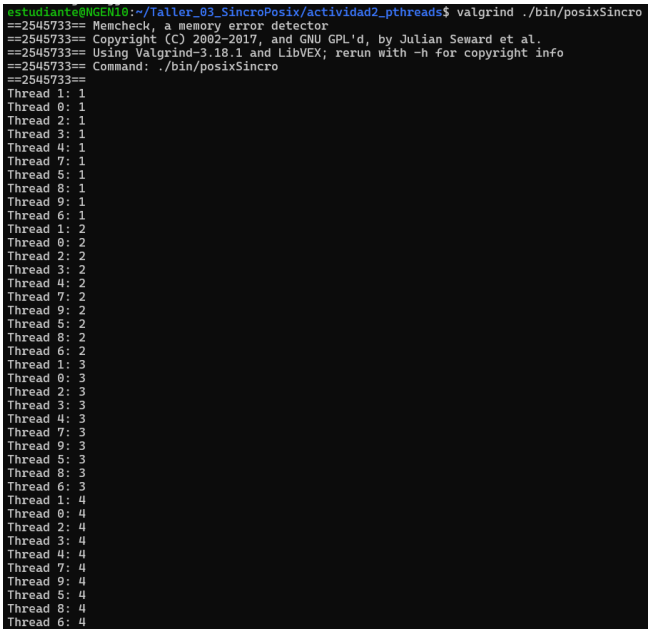
Prueba 6 – No hay escritura fuera del rango

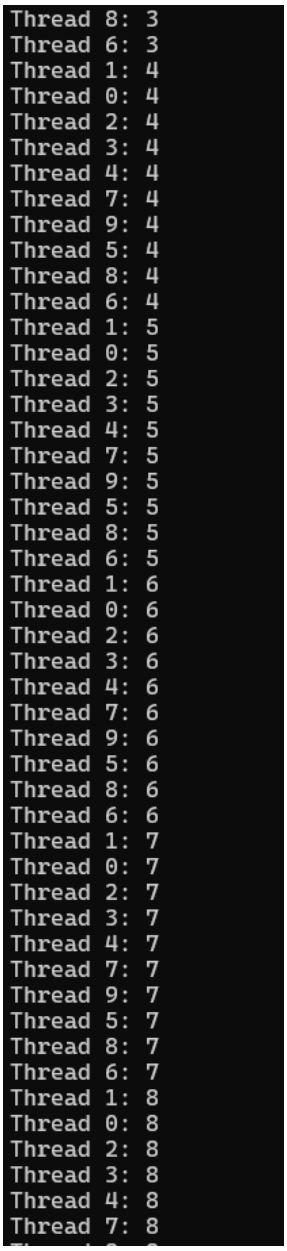
Esta prueba comprueba que el uso del búfer circular es estable y no provoca errores de memoria.

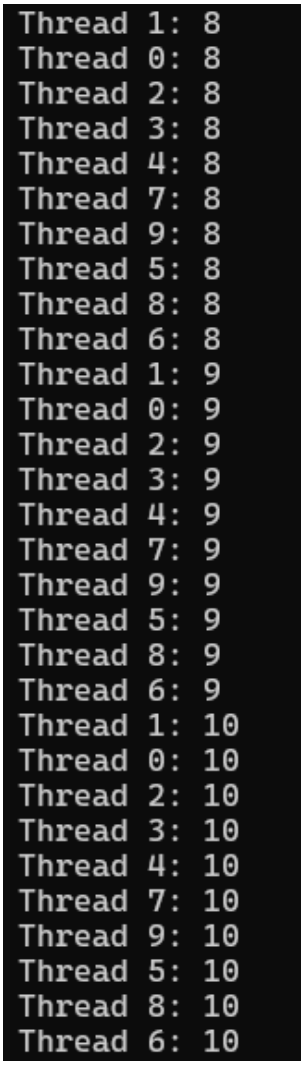
Esperado:

El programa no muestra errores de memoria.

El comportamiento observado es coherente: los mensajes se imprimen sin cuelgues ni salidas raras. Lógicamente, los índices protegidos por el mutex se mantienen siempre entre 0 y MAX_BUFFERS - 1.







```
==2545733==
==2545733== HEAP SUMMARY:
==2545733==     in use at exit: 272 bytes in 1 blocks
==2545733==   total heap usage: 19 allocs, 18 frees, 8,118 bytes allocated
==2545733==
==2545733== LEAK SUMMARY:
==2545733==    definitely lost: 0 bytes in 0 blocks
==2545733==    indirectly lost: 0 bytes in 0 blocks
==2545733==      possibly lost: 272 bytes in 1 blocks
==2545733==    still reachable: 0 bytes in 0 blocks
==2545733==         suppressed: 0 bytes in 0 blocks
==2545733== Rerun with --leak-check=full to see details of leaked memory
==2545733==
==2545733== For lists of detected and suppressed errors, rerun with: -s
==2545733== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Explicación:

- El uso del mutex asegura que los índices del búfer circular no se alteren de forma simultánea.
- El avance con módulo mantiene los índices dentro del rango permitido.

Conclusión:

El programa accede al búfer circular de manera segura y no ocurre escritura fuera de los límites.

Prueba 7 – Orden estable en la salida

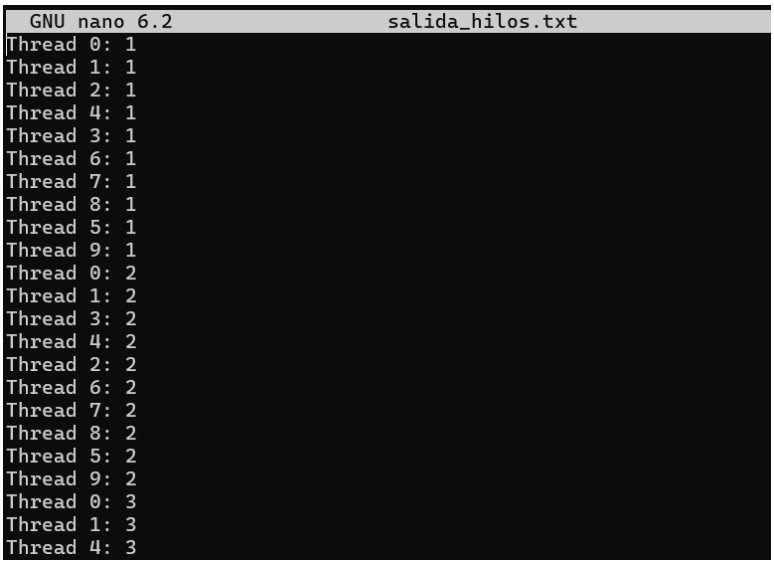
Esperado:

Cada línea tiene el formato Thread X: Y.

No se ven líneas mezcladas ni cortadas a la mitad.

El spooler imprime exactamente las líneas que los productores guardan en el búfer, sin caracteres basura.

```
estudiante@NGEN10:~/Taller_03_SincroPosix/actividad2_pthreads$ ./bin/posixSincro > salida_h
ilos.txt
estudiante@NGEN10:~/Taller_03_SincroPosix/actividad2_pthreads$ ls
bin  data  Makefile  salida_hilos.txt  src
estudiante@NGEN10:~/Taller_03_SincroPosix/actividad2_pthreads$
```



Explicación:

- El spooler imprime una línea a la vez gracias al mutex.
- Ningún hilo productor puede escribir mientras el spooler está en su sección crítica.

Conclusión:

La salida es estable, ordenada y no presenta interferencias entre hilos.

Prueba 8 – Cancelación del spooler

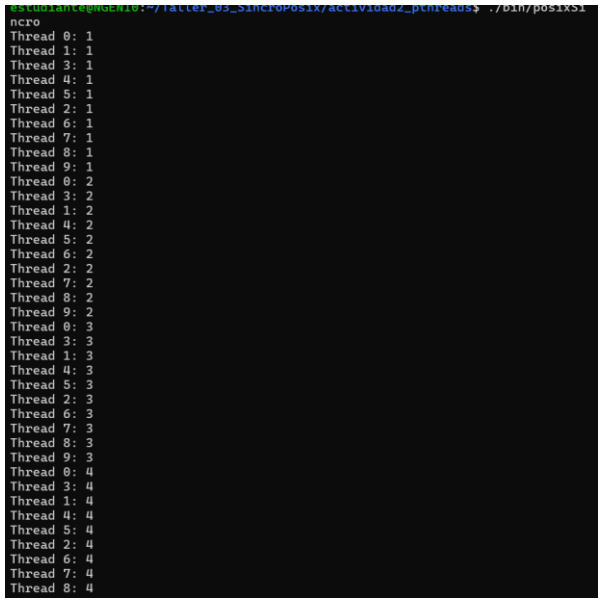
Esperado:

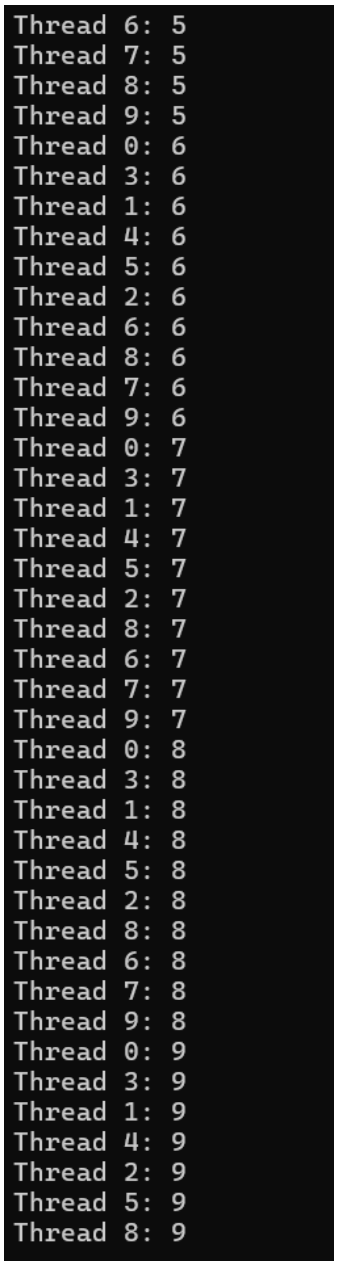
El programa termina sin quedarse bloqueado.

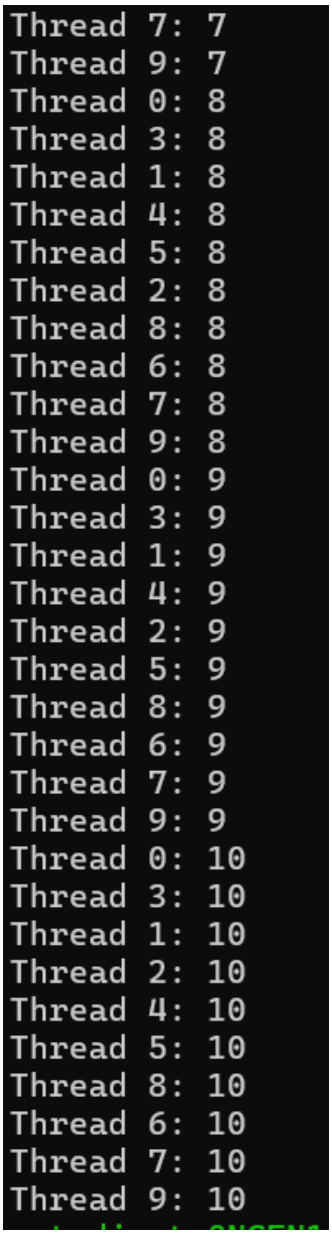
No hay deadlocks ni necesidad de matar el proceso manualmente.

No se muestran errores de pthreads al cancelar el spooler.

La ejecución finaliza de forma limpia después de que las líneas pendientes han sido impresas.







Explicación:

- El spooler permanece en un ciclo que puede ser cancelado de manera segura porque espera en una condición o entra y sale constantemente del mutex.
- Esto permite que la cancelación funcione sin riesgos.

Conclusión:

La terminación del hilo spooler es limpia, controlada y no afecta la ejecución del programa principal.

6. Conclusiones finales del taller

- Este taller permitió comprender de forma práctica la diferencia entre sincronización entre procesos (Actividad 1) y sincronización entre hilos (Actividad 2), cada una usa mecanismos apropiados del estándar POSIX.
- En la Actividad 1 se logró implementar un productor y un consumidor reales, usando memoria compartida y semáforos con nombre. Esto permitió observar cómo dos procesos distintos pueden coordinarse sin condiciones de carrera.
- En la Actividad 2 se aplicaron mutex y variables de condición para manejar múltiples hilos dentro del mismo proceso. Esto reforzó la importancia de proteger las secciones críticas y usar señales para bloquear y despertar hilos correctamente.
- La estructura modular del proyecto, con carpetas separadas por actividad y Makefiles individuales más un Makefile general, permitió organizar mejor el código y facilitó la compilación y las pruebas.

Gracias a este taller se adquirió experiencia directa en identificar, crear, sincronizar y liberar recursos del sistema operativo. Esto sienta bases sólidas para entender y desarrollar aplicaciones concurrentes más complejas.